7. Programming Policies and Guidelines

Created: April 1, 2003
Updated: September 16, 2003

Style Guide

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

Introduction

This chapter discusses the programming policies and coding styles used in the C++ Toolkit. Adherence to these guidelines will promote provide uniform coding, better documentation, easy to read code, and therefore more maintainable code.

Chapter Outline

The following is an outline of the topics presented in this chapter:

- Naming conventions, Indentation, and Bracing
    - Naming conventions
    - Name prefixing and/or the use of namespaces
    - Use of the NCBI name scope
    - Use of include directives
    - Code indentation and bracing
    - Class declaration
    - Function declaration
    - Function definition
    - Use of whitespace
    - Standard Header Template

Naming conventions, Indentation, and Bracing

"In My Egotistical Opinion, most people's C programs should be indented six feet downward and covered with dirt." -- Blair P. Houghton

Nevertheless, here goes:

- Naming conventions
  - Type Names
  - Preprocessor Define/Macro
  - Function arguments
  - Constants
  - Class and structure member variables
  - Class member functions
  - Module static functions and data
  - Global ("extern") functions and data
- Name prefixing and/or the use of namespaces
- Use of the NCBI name scope
- Use of include directives
- Code indentation and bracing
- Class declaration
- Function declaration
- Function definition
- Use of whitespace

Naming conventions

See Table 1 for naming conventions.

Table 1. Naming Conventions

Synopsis	Example
Type Names	
C*ClassTypeName*	*class CMyClass { ..... };*
I*InterfaceName*	*class IMyInterface { ..... };*
S*StructTypeName*	*struct SMyStruct { ..... };*
U*UnionTypeName*	*union UMyUnion { ..... };*

Synopsis	Example
***E**EnumTypeName*	*enum EMyEnum { ..... };*
***F**FunctionTypeName*	*typedef int (*FMyFunc)(void);*
***P**PredicateName*	*struct PMyPred { bool operator() (.... , ...); };*
***T**AuxiliaryTypedef (see table footnote)*	*typedef map<int,string> TMapIntStr;*
T**Iterator_**I	*typedef list<int>::iterator TList_I;*
T**ConstIterator_**CI	*typedef set<string>::const_iterator TSet_CI;*
***N**Namespace (see also)*	*namespace NMyNamespace { ..... }*
Preprocessor Define/Macro	
MACRO_NAME	*#define MY_DEFINE 12345*
macro_arg_name	*#define MY_MACRO(x, y) (((x) + 1) < (y))*
Function arguments and local variables	
func_local_name	*void MyFunc(int foo, const CMyClass& a_class) { size_t foo_size; int bar;*
Constants	
***k**ConstantName*	*const int kMyConst = 123;*
***e**EnumConstantName*	*enum EMyEnum { eMyEnum_1 = 11, eMyEnum_2 = 22, eMyEnum_3 = 33 };*
***f**FlagConstantName*	*enum EMyFlags { fMyFlag_1 = 0x1, // = (1<<0) fMyFlag_2 = 0x2 // = (1<<1) }; typedef int TMyFlags; // holds a binary OR of "EMyFlags"*
Class and structure data members (fields)	
***m**_ClassMemberName*	*class C { short int m_MyClassData; };*
***m**_StructFieldName*	*struct S { int my_struct_field; };*
***sm**_ClassStaticMemberName*	*class C { static double sm_MyClassStaticData; };*
Class member functions (methods)	
Func	*bool MyFunc(void);*
Module static functions and data	
***s**_StaticFunc*	*static char s_MyStaticFunc(void);*
***s**_StaticVar*	*static int s_MyStaticVar;*
Global (*"extern"*) functions and data	
GlobalFunc	*double MyGlobalFunc(void);*
***g**_GlobalVar*	*short g_MyGlobalVar;*

(*) The auxiliary typedefs(like T AuxiliaryTypedef) are usually used for an ad-hoc type mappings(especially when using templates) and not when a real type definition takes place.

Name prefixing and/or the use of namespaces

In addition to the above naming conventions that highlights the nature and/or the scope of things, one should also consider the use of prefixes (or namespaces) in order to:

- avoid name conflicts
- indicate the package which the thing belongs to

E.g. it will always be a good idea to name your new base class (for some package "`Fo""`) "***CFooObject***" rather than just "***CObject***". -- And the same is true for all other things from this package, so name the constants "`k`**`Foo`**`Someconst`", types "***TFooSometype***", etc.

It is decided that we usually do not use C++ namespaces (other than the NCBI-wide namespace "*ncbi::*") in NCBI.

One can emulate a sort of namespace by putting the things into a structure or class scope, like "*struct N***Foo***Globals { .....; void SomeFunc(int); ... };*". Then just use an explicit prefix when referencing these things, e.g. call ***"NFooGlobals::SomeFunc(5)"***. This approach can be useful to group and prefix a set of global functions. As for classes, etc. it is still better to use simple prefixes (as described above) instead.

Use of the NCBI name scope

`<ncbistl.hpp>`

All NCBI-made API must be put into the single namespace. For this reason, there are two preprocessor macros, `BEGIN_NCBI_SCOPE` and `END_NCBI_SCOPE`, that must embrace **all** NCBI C++ API code -- both declarations and definitions (see examples). Inside these "brackets", all *"std::"* and *"ncbi::"* scope prefixes can(and must!) be omitted. Safely.

For the code that does not define new API but merely **uses** NCBI C++ API, there is a macro `USING_NCBI_SCOPE` (semicolon-terminated) that makes all "std"- and "ncbi"-related types and protos be visible by default, without the explicit scope specification (*"std::"* and *"ncbi::"* prefixes, respectively).

Use macro `NCBI_USING_NAMESPACE_STD` (semicolon-terminated) if you want make visible only "std"-related code, without the "ncbi"-related one.

Use of include directives

Unless there is a file in the local directory that must be included and that is not on the INCLUDE path, you must always use the following form of the #include directive:

#include <foo.hpp>

Use the *#include "foo.hpp"* form only when a file in the local directory must be included. In general, if an include file is commonly used, it must be in one of the directories in the INCLUDE path, in which case the *#include <foo.hpp>* form should be used.

Code indentation and bracing

4-space indentation only! Tabulation symbol **must not** be used for indentation.

Try not to cross the "standard page boundary" of **80** symbols.

In *if, for, while, do, switch, case*, etc. and type definition statements:

```
if (...) {
    .....;
} else if (...) {
    .....;
} else {
    .....;
}

for (...;  ...;  ...) {
    .....;
}

while (...) {
    .....;
}

do {
    .....;
} while (...);

switch (...) {
case ...: {
    .....;
    break;
}
} // switch

struct|union|enum <[S|U|E]TypeName> {
    .....;
};

class
{
    .....;
};

try {
    .....;
}
catch (exception& e) {
    .....;
}
```

Class declaration

```
class CFooClass
{
public:
    // Constructors and Destructor (they are usually public)
    CFooClass(bool init_bool = true);
    CFooClass(int  init_int);
    ~CFooClass(void);

    // Members and Methods
    float PublicFunc(void);
    // (NOTE:  the use of public data members is
    //         strictly discouraged!)
    int   m_PublicData;

protected:
    double m_ProtectedData;
    static int ProtectedFunc(char ch);

private:
    int    m_PrivateData;
    double PrivateFunc(int some_int = 123);

    // Friends
    friend bool  SomeFriendFunc(void);
    friend class CSomeFriendClass;
private:
    // Prohibit default initialization and assignment
    // -- e.g. when the member-by-member copying is
    // dangerous.
    // (These two methods are not even implemented.)
    CFooClass(const CFooClass&);
    CFooClass& operator= (const CFooClass&);
};
```

Function declaration

```
// Explain here what MyFunc1() does, and what it returns
int MyFunc1(void);

// Explain here what MyFunc2() does, and what it returns
bool MyFunc2
(double  arg1,      // short description of "arg1"
string& arg2,      // short description of "arg2"
long    arg3 = 12  // short description of "arg3"
);
```

Function definition

```
bool MyFunc2
(double  arg1,
string& arg2,
long    arg3)
{
    .......
    .......
    return true;
}


// For static function, put all comments on what the
// function does and what it returns right here, at
// the point of the function definition
static long s_MyFunc3(void)
{
    .......
    .......
}
```

Use of whitespace

As the above examples do not make all of our policies on whitespace clear, here are some explicit guidelines:

- When reasonably possible, use spaces to align corresponding elements vertically. (This overrides most of the rules below.)
- Leave one space on either side of most binary operators, and two spaces on either side of boolean *&&* and ||.
- Put one space between the names of flow-control keywords and macros and their arguments, but no space after the names of functions except when necessary for alignment.
- Leave two spaces after the semicolons in *for(...; ...; ...)*.
- Leave whitespace around negated conditions so that the *!* stands out better.
- Leave two blank lines between function definitions.

Standard Header Template

A standard header template file, *header_template.hpp*, has been provided in the *doc/standard* directory that can be used as a template for creating header files. This header file adheres to the standards outlined in the previous sections and uses a documentation style for files, classes, methods, macros etc. that allows for automatic generation of documentation from the source code. It is strongly suggested that you obtain a copy of this file and model your documentation using the examples in that file.

Guidelines

Introduction to some of C++ and STL features and techniques

C++ Implementation Guide

Use of STL (Standard Template Library)

Use the Standard Template Library(STL), which is part of ANSI/ISO C++. It'll make programming easier, as well as make it easier for others to learn and maintain your code.

Use of C++ exceptions

Exceptions are useful. However, since exceptions unwind the stack, you must be careful to destroy all resources (such as memory on the heap and file handles) in every intermediate step in the stack unwinding. That means you must always catch exceptions, even those you don't handle, and delete everything you are using locally. In most cases it's very convenient and safe to use the ***auto_ptr*** template to ensure the freeing of temporary allocated dynamic memory for the case of exception.

Design

Use abstract base classes. This increases the reusability of code. Whether a base class should be abstract or not depends on the potential for reuse.

Don't expose complex member variables, rather expose member functions that manipulate the member variables. This increases reusability and flexibility. For example, this frees you from having the string in-process -- it could be in another process or even on another machine.

Don't use multiple inheritance (i.e. *class A: public B, public C {}*) unless creating interface instead of implementation. Otherwise, you'll run into all sorts of problems with conflicting members, especially if someone else owns a base class. The best time to use multiple inheritance is when a subclass multiply inherits from abstract base classes with only pure virtual functions.

Some people prefer the Unified Modelling Language to describe the relationships between objects.

C++ Tips and Tricks

Writing something like *map<int, int, less<int>>* will give you weird errors; instead write *map<int, int, less<int> >*. This is because >> is reserved word.

Do use pass-by-reference. It'll cut down on the number of pointer related errors.

Use *const* (or *enum*) instead of *#define* when you can. This is much easier to debug.

Header files should contain what they contain in C along with classes, const's, and in-line functions (in-line functions are functions defined within the braces of the class or declared *inline*. In-line functions must be in header files as compilers need to see the source in order to inline properly.

g++ might not build shared libraries properly on Solaris without -mimpure-text.

See the C++ FAQ

Standard template library (STL)

The STL is a library included in ANSI/ISO C++ for stream, string, and container(linked lists, etc.) manipulation.

STL tipsandtricks

end() does not return an iterator to the last element of a container, rather it returns a iterator just beyond the last element of the container. This is so you can do constructs like

```
for (iterator = container.begin();
iterator != container.end(); iterator++)
```

If you want to access the end element, use "*--container.end()*".

Iterator misuse causes the same problems as pointer misuse. There are versions of the STL that flag incorrect use of iterators.

Iterators are guaranteed to remain valid after insertion and deletion from list containers, but not vector containers. Check to see if the container you are using preserves iterators.

If you create a container of pointers to objects, the objects are not destroyed when the container is destroyed, only the pointers are. Other that maintaining the objects yourself, there are several strategies detailed in the literature.

If you pass a container to a function, don't add a local object to the container. The local variable will be destroyed when you leave the function.

When using Solaris make, the macro `.KEEP_STATE` can be used to track hidden dependencies, but this doesn't work well with `Sunpro C++ 5.0` and STL. The reason is because the STL headers include files with names of the form **.cc*. When make sees these files, it tries to compile them using the implicit makefile rules.One workaround is to redefine the implicit rules to ignore **. cc*:

```
.SUFFIXES:
.SUFFIXES: .o .c .h .hpp .cpp
.cpp.o:
$(CCC) $(CFLAGS) -c $<
and run make with "make -r"
```

In SunPro environment, use `"CC -xar ..."` rather than `"ar ..."` to compose a library that defines template-based functions and classes for external use.

When declaring a map, use the form *map<string, string, less<string>* >. The third argument isn't necessary on most compilers, but SunPro 4.2 is a bit old and needs it because it doesn't do default values for templates.

SunPro 4.2 doesn't understand *"using namespace std;"*, and it was decided to prohibit the explicit use *"using namespace XXX;"* in NCBI C++ code. For the use of standard C++ template classes(including STL) and/or NCBI namespaces click here.

The ***basic_string*** class has a few pitfalls

C++/STL pit-falls and discouraged/prohibited features

- STL and standard C++ library's bad guys
    - Old vs. new stream classes

    - Non-standard classes

- C++ bad guys

    - Operation overload
    - Assignment and copy constructor overload
    - Namespaces
    - Omitting "void" in a no-arg function proto
    - Do not mix malloc and new

STL and standard C++ library's bad guys

Old vs. new stream classes

#include <corelib/ncbistre.hpp> [also included in <*corelib/ncbistd.hpp*>]

Do not use the new iostream library included with Sunpro C++ 5.0 early access release refresh. It is extremely buggy. Instead we will use the older version of iostream.h by default.

It was a big question if we should use "old-fashioned" C++ stream classes (from *#include <iostream.h>*, etc.) or "templated" ones (from *#include <iostream>*, etc.). Some platforms do not support "new" streams yet; and although there is a way to import most STL classes and string template class using package like STLport, however C++ stream classes usually cannot be implemented formally (yet in full) by a third party because of their close interaction with the underlying file system.

The solution is to use an *"intersection"* between the new and old stream functionality, and thus allow one to use either old or new streams depending on whether the given platform supports new streams or not. Thus, for the reason of the "old/new streams" portability, one **must** use:

- *#include <corelib/ncbistre.hpp>* [also included in <*;corelib/ncbistd.hpp*>]

and **must never** do directly:

- *#include <iostream>*, etc.
- *#include <iostream.h>*, etc.

Then, **always** use:

- ***"CNcbiIstream", "CNcbiOstream"***, etc.

and **never** use:

- ***"istream", "ostream"***, etc. or even
- *"::istream", "::ostream"*, etc. or *"std::istream", "std::ostream"*, etc.

and, of course, the use of stream iterators and "***stringstream***"-related classes (from ***<sstream>***) is not allowed as well.

By following this approach, we should be able to use both "new" and "old" stream libs now "for free". The needed code discipline will be maintained "automagically" as any "old streams"-only and "new streams"-only code would not compile immediately. On the other hand, we reserve a possibility to replace "*typedef*"s by derived classes with added functionality (e.g. in order to extend the new/old stream API intersection area, or to add some features like overridden ***overflow()***/***underflow()*** callbacks).

If compiler supports both "old" and "new" C++ streams then "old-fashioned" C++ libs will be engaged by default. This can be alternated by *#defin*'ing preprocessor variable "`NCBI_USE_NEW_IOSTREAM`".

Non-standard STL classes

Don't use ***hash*** or the ***rope*** classes from some versions of the STL. These are non-standard additions. If you have questions about what is/isn't in the standard library, consult the C++ standards.

C++ bad guys

Operation overload

Do not use operator overloading for the objects where they have unnatural or ambiguous meaning. E.g. the defining of operation "==" for your class ***"CFoo"*** so that there exist { CFoo a,b,c; } such that *(a == b)* and *(b == c)* are *true* while *(a == c)* is *false* would be a very bad idea. It turns out that otherwise, especially in large projects, people have different ideas of what an overloaded operator means, leading to all sorts of bugs.

Assignment and copy constructor overload

Be advised that the default initialization *{CFoo foo = bar;}* and assignment *{CFoo foo; ...; foo = bar;}* do a member-by-member copying. This is not suitable and can be dangerous sometimes. And if you decide overwrite this default behaviour by your own code like:

```
class CFoo {
    // a copy constructor for initialization
    CFoo(const CFoo& bar) { ... }
    // an overloaded assignment(=) operator
    CFoo& operator=(const CFoo& bar) { if (&bar != this) ... }
};
```

it is **extremely important** that:

- **both** copy constructor and overloaded assignment be defined

- they have **just the same** meaning that is *{CFoo foo = bar;}* is equivalent to *{CFoo foo; foo = bar;}*
- there is a check for a self-assignment case in your overloaded assignment operation

In many cases when you dont want to have the assignment and copy constructor at all, just add to your class something like:

```
class CFoo {
    .............................
private:
    // Prohibit default initialization and assignment
    CFooClass(const CFooClass&) { _TROUBLE; }
    CFooClass& operator=(const CFooClass&) { _TROUBLE; return *this;}
};
```

Namespaces

Do not define your own namespaces at all. Instead, if you want to ensure that you will not get into a name conflict, see the workaround (quasi-namespace) technique. Do not engage foreign namespaces (with *"using namespace XXX;"*) unless this is practically unavoidable. For the use of standard C++ template classes(including STL) and/or NCBI namespaces click here.

Omitting "void" in a no-argfunctioproto

Do not omit *"void"* in prototype of a function without arguments (e.g. always write ***"int f(void)"*** rather than just ***"int f()"***).

Do notmix mallocandnew

On some platforms, malloc and new may use completely different memory managers, so never "free()" what you created using "new" and never "delete" what you created using "malloc()". Also, when calling C code from C++ **always** allocate any structs or other items using "malloc()". The C routine may use "realloc()" or "free()" on the items, which can cause memory corruption if you allocated using "new."